TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Programmierwerkzeuge

Übersicht I

Die Erstellung von großen Software-Systemen (Implementierungsphase) ist komplex.

- Viele Entwickler arbeiten gleichzeitig an der Software
- Die Software besteht aus vielen Dateien
- Teile der Software wird bei vielen Projekten benötigt (z.B. Parser, Email versenden, Warteschlangen)

Entwicklungswerkzeuge helfen bei der Entwicklung großer Software-Systeme. Sie helfen dem Programmierer, die Komplexität zu beherrschen.

Übersicht II

In größeren Software-Projekten fallen mehrere Aufgaben an, die nacheinander ausgeführt werden müssen, um die Software zu erstellen.

Build-Systeme helfen dabei, diese Schritte zu automatisieren.

Die Dateien des Projekts müssen verwaltet werden, Änderungen sollen zurückverfolgbar sein.

Versionsverwaltungssysteme können helfen, diese Transparenz und Verteilung zu erreichen.

Diese Tools haben verschiedene Benutzerschnittstellen, um sie zu steuern.

Integrierte Entwicklungsumgebungen ermöglichen es, die Entwicklungswerkzeuge über eine einheitliche Oberfläche zu steuern.

Build-Systeme

Build-Systeme I

In realen Projekten kann das Erstellen eines Softwaresystems ein komplexer Ablauf sein:

- viele Dateien mit Code
- unter Umständen Generierung von Teilen des Codes (z.B. Scanner, Parser)
- Erstellung zusätzlicher Artefakte (z.B. Dokumentation, JAR-Datei)
- Abhängigkeiten zwischen den auszuführenden Schritten

Build-Systeme II

Nehmen wir an, wir haben ein Projekt, welches eine Eingabe in einer bestimmten Sprache nimmt. Für diese Sprache haben wir Scanner und Parser generiert.

Wird die Sprache geändert, die man in einem Projekt übersetzen möchte, müssen die folgenden Schritte in der richtigen Reihenfolge durchgeführt werden:

- Neugenerierung des Scanners (wegen neuen Tokens)
- Neugenerierung des Parser (wegen neuen Teilen der Grammatik)
- Übersetzen aller geänderter Java-Quellen (inklusive des generierten Codes)

Vergisst man, diese Schritte (in der richtigen Reihenfolge) auszuführen, kann dies zu unerwarteten Fehlern führen (z.B. veralteter Scanner mit neuerstelltem Parser).

Build-Systeme: Ant

Apache Ant ist ein Java-spezifisches Build-System.

Es unterstützt die Automatisierung des Erstellungsprozesses von Java-Projekten.

Einzelne Aufgaben und ihre Verknüpfung werden in einer XML-Datei (build.xml) definiert.

Ant bietet einige vordefinierte Aktionen (z.B. `mkdir`, `delete`, `javac`, `java`, `jar`).

Beispiel: Ant

```
<project name="hello-world" default="compile">
  <property name="src.dir" location="src"/>
  <property name="build.dir" location="build"/>

  <target name="init">
    <mkdir dir="${build.dir}"/>
  </target>

  <target name="compile" depends="init"
          description="compile source">
    <javac srcdir="${src.dir}" destdir="${build.dir}"/>
  </target>

  <target name="clean" description="clean up">
    <delete dir="${build.dir}"/>
  </target>
</project>
```

Weitere Beispiele für Build-Systeme

Build-Systeme für Java:

- Maven (http://maven.apache.org/)
- Gradle (http://www.gradle.org/)

Build-Systeme für andere Sprachen:

- Make (meist C/C++, aber auch z.B. LaTeX)
- sbt (simple build tool; Scala)

Bemerkungen

Einige Build-Systeme unterstützen die selektive Neuerstellung. Neue und geänderte Dateien werden neu übersetzt, aktuelle Dateien bleiben erhalten.

- Analyse der Abhängigkeiten zwischen den Quelldateien ist komplex
- Abhängig von der Programmiersprache (z.B. nicht von Make unterstützt)
- Funktioniert nicht immer zuverlässig

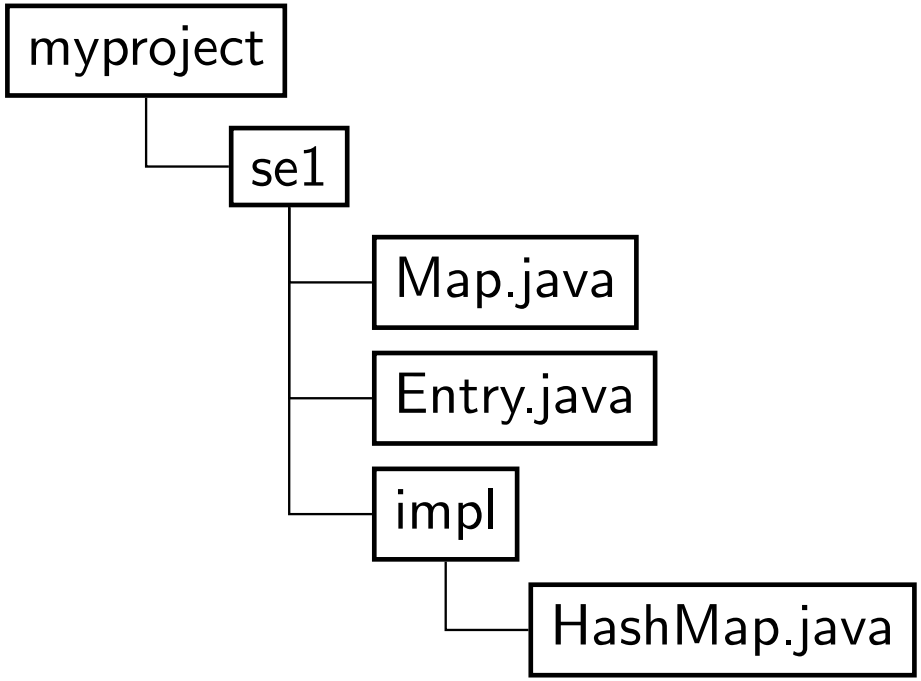
Einschub: Pakete

Java Pakete I

Sollen Klassen von anderen verwendet werden, so müssen die Namen der Klassen eindeutig werden. Wichtig ist dies vor allem bei sehr allgemeinen Begriffen wie `Entry` und `Value`.

Java bietet dazu das Konzept der Pakete (engl. *packages*) an. Die `.java`-Dateien werden in Unterordnern abgelegt und am beginn der Datei wird mit `package` deklariert, in welchem Paket sich die Klasse befindet.

Java Pakete II



Map.java:

```
package se1;

public class Map ...
```

Entry.java:

```
package se1;

public class Entry ...
```

HashMap.java:

```
package se1.impl;

public class HashMap ...
```

Java Pakete III

Auf Klassen aus anderen Paketen kann man zugreifen

- indem man sie importiert (z.B. `import se1.Map;`) oder
- über den voll qualifizierten Namen (z.B. `se1.Map m = new se1.impl.HashMap();`)

In den Namen der Pakete werden die Schrägstriche durch Punkte ersetzt.

Häufig richten sich Paketnamen nach der URL der Firma, die die Software erstellt hat (z.B. `com.google.gson`)

Die Ordnerstruktur wird vom Java-Compiler verwendet, um die richtigen Quelldateien zu eine Klasse zu finden. So wird die Klasse `se1.impl.HashMap` im Ordner `se1/impl` in der Datei `HashMap.java` gesucht.

Verwaltung von Abhängigkeiten

Bibliotheken in Java I

Wenn Java Programme ausgeliefert werden sollen, müssen sie in eine eigenständige Einheit verpackt werden.

Das Gleiche gilt für eigene implementierte Bibliotheken (siehe Abschnitt 6).

Das Standard-Format für Java ist das Java-Archiv-Format (häufig auch JAR-Datei genannt).

JAR-Dateien sind ZIP-Archive, die eine Beschreibung des Java-Programms in Form einer Manifest-Datei enthalten.

Bibliotheken in Java II

Manifest-Dateien können unter Anderem folgende Informationen enthalten:

- Angabe der Klasse mit der main-Methode (`Main-Class:`)
- Hinzufügen zusätzlicher Bibliotheken zum Classpath (`Class-Path:`)

Verwendete Bibliotheken müssen dem Java-Compiler mit der Classpath-Option mitgeteilt werden (z.B. `javac -cp MyArchive.jar`).

Außerdem müssen die Bibliotheken bei der Ausführung des Programms an den Java-Interpreter übergeben werden (z.B. `java -cp MyArchive.jar`).

Bibliotheken in Java III

Java-Archive können auf der Kommandozeile mit dem Befehl
`jar cvfm myarchive.jar mymanifest -C myfolder` erzeugt werden.

Ant bietet Unterstützung für die Erstellung von JAR-Dateien in Form einer Aktion `jar`.

```
<jar destfile="myarchive.jar" basedir="${build.dir}">
  <manifest>
    <attribute name="Main-Class"
          value="mypackage.MyMainClass"/>
  </manifest>
</jar>
```

Ausgeführt wird das so erzeugte Java-Archiv mit `java -jar myarchive.jar`.

Unter Windows kann bei richtig installiertem Java ein Java-Archive mit Dopppelklick wie eine native Windows Anwendung ausgeführt werden.

Verwaltung von Abhängigkeiten

In realen Projekten werden häufig eine große Menge an externen Bibliotheken verwendet.

Verwaltung der Abhängigkeiten sehr mühsam:

- Bibliotheken haben andere Bibliotheken als Abhängigkeit
- genaue Version der Bibliotheken wichtig
- Integration in der Erstellungsprozess nötig

Früher: Viele JAR-Dateien in einem Ordner und manuelle Angabe der Dateien (z.B. beim Kompilieren)

Heute: Integration in Build-Systeme

Apache Ivy

Apache Ivy ist ein Abhängigkeitsverwalter (dependency manager) für Ant.

Es bietet:

- Zugriff auf zentrale Repositories von Bibliotheken (z.B. Maven Central)
- automatische Verwaltung der JAR-Dateien (Finden im Repository, Herunterladen und Ablage auf der Festplatte)
- Integration der benötigten Bibliotheken in den Erstellungsprozess (z.B. Angabe der JAR-Dateien beim Kompilieren)

Integration in Ant

Die Abhängigkeiten des Projekts werden in einer separaten XML-Datei (ivy.xml) angegeben.

- Angaben über das aktuelle Projekt (Organisation, Modulname)
- Angabe der Abhängigkeiten (Organisation, Modulname und Version)

Integration in Ant

Ivy definiert neue Ant-Aktionen, die in Ant-Tasks verwendet werden können.

`ivy:retrieve`: Liest die XML-Konfiguration (ivy.xml) und lädt die Abhängigkeiten in ein Verzeichnis `lib` herunter.

`ivy:report`: Erstellt eine Übersicht über die Abhängigkeiten des Projektes.

Die Integration der heruntergeladenen JAR-Dateien erfolgt mit Standard Ant-Aktionen (Definition eines classpaths)

Beispiel Ivy: ivy.xml

```
<ivy-module version="2.0">
  <info organisation="de.uni-kl" module="hello-world"/>
  <dependencies>
    <dependency org="org.apache.commons" name="commons-email"
                rev="1.3.2" />
    <dependency org="com.google.code.gson" name="gson"
                rev="2.5" />
  </dependencies>
</ivy-module>
```

Beispiel Ivy: build.xml

```
...
<property name="lib.dir" location="lib"/>

<path id="lib.path.id">
  <fileset dir="${lib.dir}"/>
</path>

<target name="resolve"
        description="retrieve dependencies with ivy">
  <ivy:retrieve/>
</target>

<target name="compile" depends="init,resolve"
        description="compile source">
  <javac srcdir="${src.dir}" destdir="${build.dir}"
    classpathref="lib.path.id"/>
</target>
...
```

Bemerkungen

Auch andere Build-Systeme haben Abhängigkeitsverwaltung integriert (z.B. Maven, Gradle und sbt)

Neu hinzugefügte Abhängigkeiten werden beim nächsten Bauvorgang heruntergeladen; Dateien, die aktuell sind, werden nicht erneut heruntergeladen.

Es sind noch viel komplexere Konfigurationen möglich, unter anderem auch die Veröffentlichung der Software auf einem Server.

Versionsverwaltungssysteme

Problemstellung

Bei der Entwicklung großer Softwaresysteme im Team stellen sich neue Herausforderungen:

- Code und Änderungen müssen für alle Teammitglieder zugänglich sein
- Dateien müssen auf einem Server gesichert werden
- Historie der Änderungen sollte erhalten bleiben

Ein Versionsverwaltungssystem (VCS; version control system) kann helfen, die Probleme zu lösen.

Git I

Git ist ein verteiltes Versionsverwaltungssystem (DVCS; distributed version control system). Eine *Version* ist ein klar umrissener Zustand des Inhalts eines Ordners.

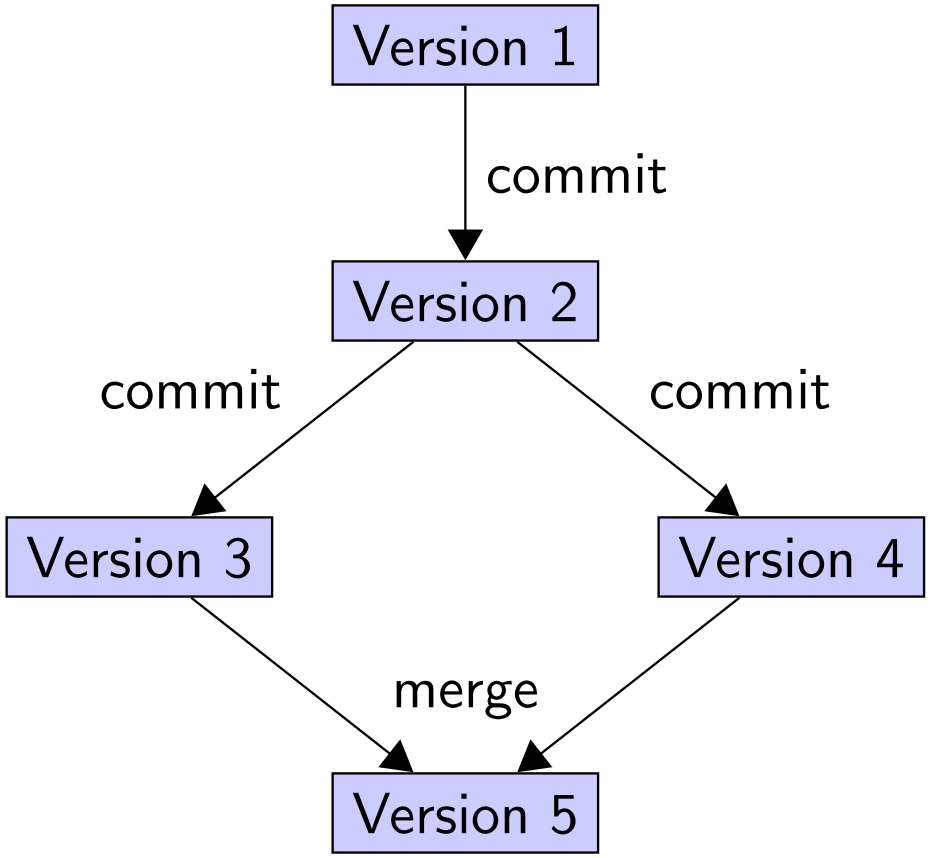
Die Verwaltung von Versionen erfolgt lokal in einem *Repository*, es kann jedoch mehrere Kopien dieses Repositories auf unterschiedlichen Rechnern geben.

Wir werden uns ansehen,

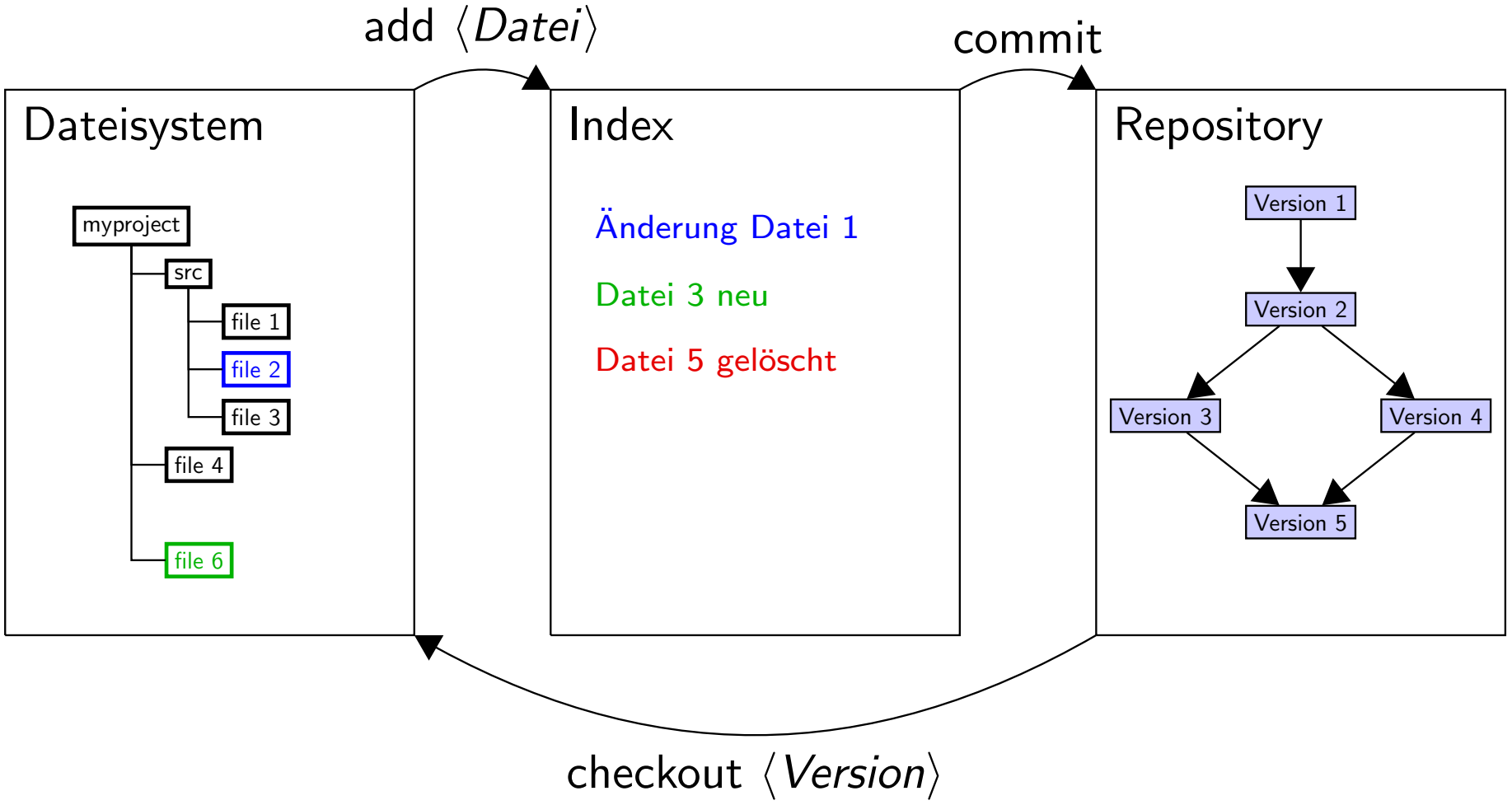
- wie neue Versionen erstellt werden (commit),
- wie Konflikte entstehen und wie sie beseitigt werden und
- wie Änderungen zwischen den Mitgliedern eines Teams ausgetauscht werden können.

Git II

- Ein *Commit* von Änderungen erzeugt neue Version auf Basis einer bestehenden
- Parallele Änderungen führen zu mehren Kindversionen (hier Versionen 3 und 4)
- Ein *Merge* führt zwei Versionen zu einer zusammen
- Eine Übersicht über die aktuellen Versionen können als *Log* angezeigt werden.



Git III


add ⟨Datei⟩
commit
Dateisystem
myproject
src
file 1
file 2
file 3
file 4
file 6
Index
Änderung Datei 1
Datei 3 neu
Datei 5 gelöscht
Repository
Version 1
Version 2
Version 3
Version 4
Version 5
checkout ⟨Version⟩

Git IV

- Beim Merge kann es zu *Konflikten* kommen, die manuell gelöst werden müssen
- Merge Tools helfen beim Beheben dieser Konflikte, indem sie die Änderungen beider Versionen nebeneinander anzeigen.
- Ziel ist, eine konsistente neue Version zu erzeugen, die (möglichst) alle Änderungen berücksichtigt.

Git V

Beispiel eines Merge-Konflikts

```
int x = parseInt(args[0]);
int y = parseInt(args[1]);
int z = x + y;
```

```
int eing1 = parseInt(args[0]);
int eing2 = parseInt(args[1]);
int summe = eing1 + eing2;
```

```
int x = parseInt(args[0]);
int y = parseInt(args[1]);
System.out.println(x + y);
```

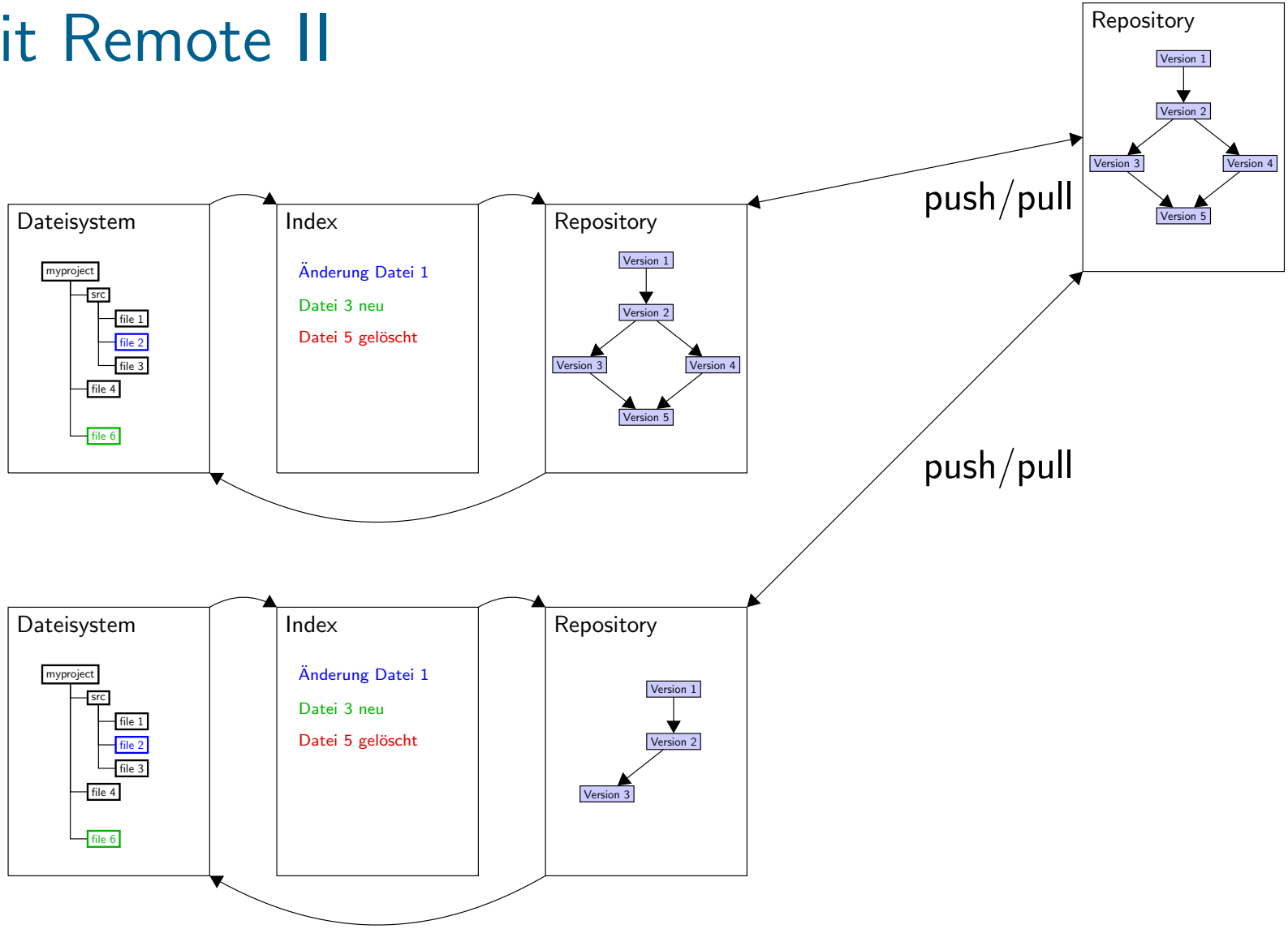
```
int eing1 = parseInt(args[0]);
int eing2 = parseInt(args[1]);
System.out.println(eing1 + eing2);
```

Git Remote I

Bis zu diesem Punkt haben wir besprochen, wie Git Versionen lokal verwaltet. Die Änderungen können jedoch auch ausgetauscht werden:

- Änderungen könnnen in eine Datei gespeichert, versendet und in einem anderen Repository angewendet werden (`git diff` und `git apply`)
- Git kann mit einem Server kommunizieren, um das lokale mit einem externen Repository abzugleichen (`git push` und `git pull`)

Git Remote II



Git Remote III

Typisches Szenario: Das Hauptrepository liegt auf einem zentralen Server.

Neue Teammitglieder können sich eine Kopie des Repositories lokal ziehen (`git clone`). Zugriff bekommt man über eine URL, die bekannt sein muss.

Alle Teammitglieder arbeiten lokal, neue Versionen werden auf den Arbeitsrechnern angelegt. Zum Austauschen dieser Versionen werden die Repositories mit dem Hauptrepository synchronisiert (`git push` und `git pull`).

Versionen von anderen Teammitgliedern müssen unter Umständen nach dem Übertragen gemergt werden, um eine eindeutige neueste Version zu erhalten.

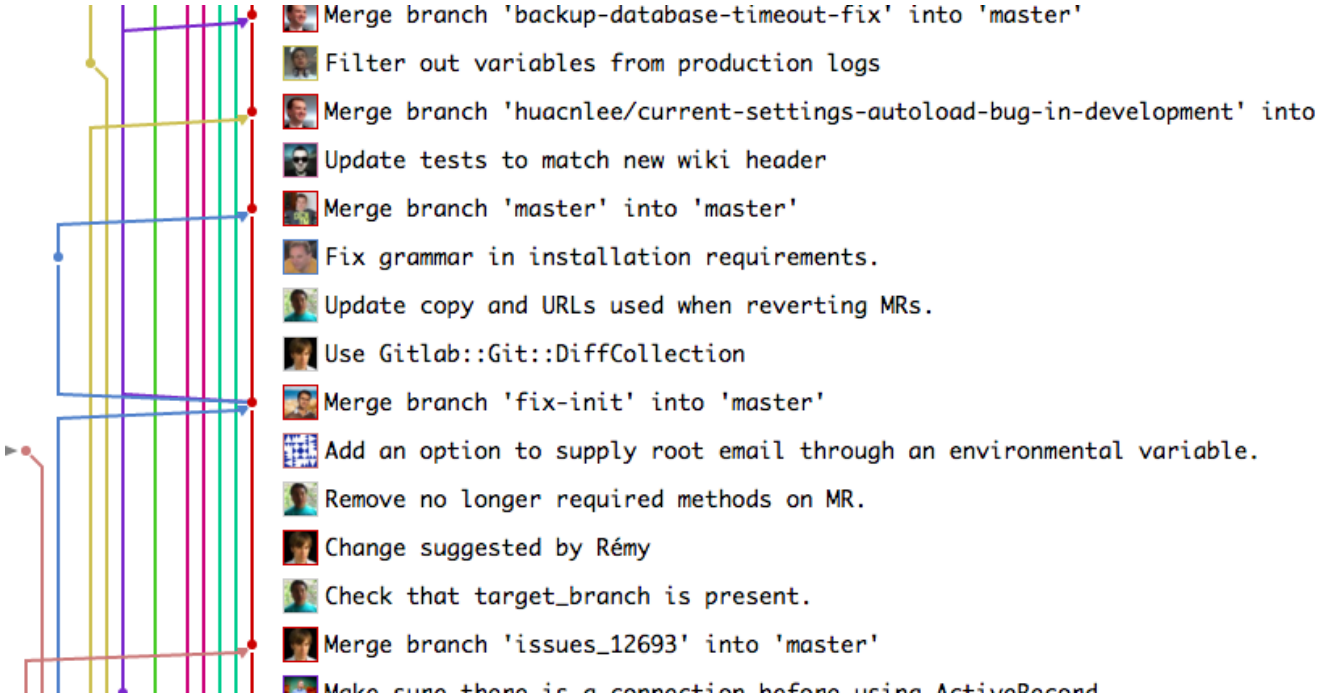
Erweiterte Git Nutzung I

Für die Verwaltung von Hauptrepositories bieten sich Webseiten wie GitHub[1] und GitLab[2] an.

- Zugriffsrechte können über eine graphische Oberfläche verwaltet werden.
- Anzeige der Versionen und des Logs der Änderungen
- Integration weiterer Dienste wie Bug-Tracker und Wikis

Erweiterte Git Nutzung II

Graphische Anzeige des Logs:



Erweiterte Git Nutzung III

Commit 9a99d8e49dc07faaaa2fae436423e11dab5a7d7e

Download as ▾ | Browse Files

Authored by **Yorick Peterse** about 23 hours ago

1 parent 7322c5a0

master

build: passed

Cache various Repository Git operations

```
This caches the output of the following methods:

* Repository#empty?
* Repository#has_visible_content?
* Repository#root_ref

The cache for Repository#has_visible_content? is flushed whenever a
commit is pushed to a new branch or an existing branch is removed.
The cache for Repository#root_ref is only flushed whenever a user
changes the default branch of a project. The cache for Repository#empty?
is never explicitly flushed as there's no need for it.
```

Changes 6 | Builds 18

Showing **6 changed files** with **140 additions** and **6 deletions**

Inline | Side-by-side

CHANGELOG — View file @9a99d8e

```
1  1  Please view this file on the master branch, on stable branches it's out of date.
2  2
3  3  v 8.5.0 (unreleased)
   4  +  - Cache various Repository methods to improve performance (Yorick Peterse)
4  5     - Ensure rake tasks that don't need a DB connection can be run without one
```

Erweiterte Git Nutzung IV

Git kann auch zur Analyse von Fehlern verwendet werden.

Annahme: Eine alte Version läuft, die neueste Version zeigt einen Fehler.

Frage: Welche Änderung hat den Fehler hervorgerufen?

⇒ Binärsuche auf den Versionen mit `git bisect`

Erweiterte Git Nutzung V

Binärsuche mit Hilfe von `git bisect`

- Starte den Vogang mit `git bisect start`
- Wechsle zu einer "schlechten" Version mit Fehler (z.B. die letzte Version)
- Teile Git mit, dass die Version "schlecht" ist mit `git bisect bad`
- Wechsle zu einer "guten" Version ohne Fehler mit `git checkout`
- Teile Git mit, dass die Version "gut" ist mit `git bisect good`

Git wechselt zur nächsten Version und aktualisiert das Dateisystem entsprechend. Mit `git bisect good` oder `git bisect bad` signalisieren Sie, ob der Fehler auftritt. Zum Schluss teilt Git Ihnen mit, welche Änderung den Fehler hervorgerufen hat.

[1] https://github.com/
[2] https://about.gitlab.com/

Git best-practices I

Welche Dateien sollen in einem Git Repository verwaltet werden, welche nicht?

- Selbst geschriebene, nicht generierte Dokumentationen wie z.B. README-Dateien sollten in Versionen verwaltet werden
- Source-Code sollte verwaltet werden
- Ergebnisse von Übersetzungsvorgängen sollten *nicht* aufgenommen werden (z.B. `.class`-Dateien)

Git bietet die Möglichkeit, bestimmte Typen von Dateien zu ignorieren. Dazu können diese in die Datei `.gitignore` eingetragen werden.

Git best-practices II

In der Datei `.gitignore` im Projektverzeichnis:

```
*.class
*.jar
out/
```

Damit werden alle `.class`-Dateien und alle `.jar`-Dateien in allen Unterordnern ignoriert. Außerdem werden alle Ordner mit Namen `out` im Projektverzeichnis ignoriert.

Übersicht wichtiger Git Kommandos

- `git init` - Neues Repository anlegen in aktuellem Ordner
- `git status` - Aktuellen Zustand des Dateisystems und Indexes in Vergleich zur letzten Version anzeigen
- `git diff` - Änderungen anzeigen
- `git add` - Änderungen in Index (staging area) aufnehmen
- `git commit` - Änderungen committen und neue Version erzeugen
- `git checkout` - Dateisystem Zustand auf Zustand einer Version setzen
- `git push` - Neue lokale Versionen zu einem entfernten Repository senden
- `git pull` - Neue Versionen von entferntem Repository holen und Dateisystem aktualisieren

Integrierte Entwicklungsumgebungen

Integrierte Entwicklungsumgebungen

Integrierte Entwicklungsumgebungen (IDEs; integrated development environments) werden häufig bei der Entwicklung von großen Softwaresystemen verwendet.

Sie bieten Hilfen beim Programmieren, vereinen jedoch auch Build-Systeme und Versionsverwaltungssysteme zu einer einheitlichen Plattform.

Hilfestellung beim Programmieren

Ein großer Vorteil gegenüber Texteditoren ist die Integration der Semantik der Programmiersprache.

Hilfestellungen sind dabei unter anderem

- Vervollständigung von Methoden, Variablen und Typen
- Navigation zur Deklartionsstelle (z.B. beim Klicken auf einen Methodennamen)
- Anzeige der erwarteten Parametertypen eines Methodenaufrufs

Hilfestellung: Codegenerierung

Viele IDEs unterstützen die Generierung von häufig implementierten Programmteilen.

- Getter/Setter zum Zugriff auf private Attribute einer Klasse
- Konstruktoren inklusive Zuweisung entsprechender Parameter
- `equals` und `hashCode` Methoden (auf Basis ausgewählter Attribute)

Refaktorierung

Refaktorierung (engl. refactoring) bezeichnet die Sturkturverbesserung von Quelltexten unter Beibehaltung des beobachteten Programmverhaltens. (Quelle: Wikipedia)

Im normalen Sprachgebrauch werden viele Änderungen, die mehrere Stellen eines Softwaresystems betreffen, als Refaktorierung bezeichnet.

Beispiele für Refaktorierungen:

- Umbenennung von Klassen und Methoden
- Verschieben von Klassen und Methoden (z.B. zwischen Paketen)
- Umwandeln eines Code-Abschnitts in eine Methode (Abstraktion)

Hinweise

IDEs helfen dabei, Refaktorierung durchzuführen. Sie bieten Assistenten, die vom Benutzer die benötigen Informationen abfragen.

Nicht alle Refaktorierungen, die von IDEs durchgeführt werden, sind tatsächlich Semantik-erhaltend. Tests helfen dabei, eventuell gemachte Fehler zu erkennen und zu beheben.

Integration von Entwicklungswerkzeugen I

Integrierte Entwicklungsumgebungen stellen Umgebungen dar, mit der sich möglichst viele Aufgaben der Softwareentwicklung zentral steuern lassen.

- semantikbewusster Texteditor
- Refaktorierung
- Integration von Build-Systemen
- Integration von Versionsverwaltungssystemen
- Debugger (zeigt Zustand des Programms zur Laufzeit)

Viele IDEs unterstützen Plugins, womit sich weitere Funktionalität nachrüsten lässt.

Integration von Entwicklungswerkzeugen II

Refaktorierung

Assistenten können per Menü oder Tastenkürzel aufgerufen werden.

Build-Systeme

Aufgaben können direkt aus der IDE gestartet werden.
Abhängigkeiten zu externen Bibliotheken werden übernommen (z.B. für Vervollständigungen)

Versionsverwaltungssysteme

Neu angelegte Klassen können direkt zum VCS hinzugefügt werden.
Commits und Updates können direkt aus der IDE heraus ausgeführt werden.

Beispiele für IDEs

IDEs für Java:

- IntelliJ IDEA (http://www.jetbrains.com/idea/)
- Eclipse (http://www.eclipse.org/)
- Netbeans (http://www.jetbrains.com/idea/)

IDEs für andere Sprachen:

- Microsoft Visual Studio (C++, C#, ...; für Windows)
- KDevelop (C, C++; für Unix)
- Apple Xcode (C, C++, Objective-C; für Mac)